DEPTH FIRST SEARCH

ISTANZA: G = (V, E)

COSA FA

1. Scopre **tutti** i vertici del grafo scoprendo prima quelli in profondità.
   Quindi $\forall$ v $\in$ V visita il sotto grafo in profondità.
2. Calcola $\forall$ v $\in$ V il ***tempo*** di ***inizio e fine scoperta*** e $\forall$ (i, j) $\in$ E etichetta gli archi in base al nodo di arrivo *(vedere proprietà sotto)*.
3. *(Principio di funzionamento)*
   Scopre ogni volta un vertice più profondo fino a raggiungere la massima profondità del ramo corrente.
4. Genera una ***foresta*** *(Foresta DFS)*, ovvero un insieme di alberi disgiunti.
   FORESTA DFS
   Denotiamo la foresta DFS con **G_π = (V, E_π)** tale che:
   V è l'insieme dei vertici del grafo originario, poiché DFS visita tutti i nodi.
   E_π rappresenta gli archi della foresta e lo definiamo come:
   E_π = { (v.π, v) | v $\in$ V AND v.$\pi \neq$ NIL }

ALGORITMO

Al fine di evitare di eseguire dei loop e di evitare il rischio di lasciare alcuni nodi inesplorati, l'algoritmo memorizza alcune informazioni su ogni nodo:
COLORE

- WHITE: ancora non scoperto
- GREY: vertice scoperto ma non analizzato completamente
- BLACK: vertice completamente analizzato

PREDECESSORE
v.π - predecessore di *v* nella visita DFS, il vertice che mi ha portato a *v*.

TEMPI

- v.d - tempo di inizio scoperta di *v*, passo nel quale lo scopro
- v.f - tempo di fine scoperta di *v*, passo nel quale ho finito di esplorarlo
  (ovviamente v.d < v.f)

L'algoritmo inoltre si divide in due procedure, la prima di **inizializzazione** e che avvia la visita vera e propria, mentre la seconda si occupa di visitare effettivamente i nodi.

INIZIALIZZAZIONE
Tempo $\Theta(|V|)$ poiché inizializza tutti i vertici del grafo.
DFS(G)
for all v $\in$ V
   v.col = WHITE
   v.π = NIL
time = 0

for all v $\in$ V
   if v.col == WHITE
      DFS_Visit(G, v)

VISITA
Tempo $\Theta(|E|)$ poiché per ogni vertice *v* analizza tutta la sua Adj[*v*] e $\sum \limits_{i = 0}^{|V|} |Adj[i]| = |E|$

```
DFS_Visit(G, u)
time++
u.d = time
u.col = GREY

for all w ∈ Adj[u]
    if w.col == WHITE
        w.π = u
        DFS_Visit(G, w)

u.col = BLACK
time++
u.f = time
```

PROPRIETÀ

- **Teorema delle parentesi:** ∀ u, v ∈ V, con A = [d[u], f[u]] e B = [d[v], f[v]] gli **unici** casi possibili sono:
  - A $\cap$ B = $\empty$ *(B viene scoperto e finisce o prima o dopo A)*
  - A $\subseteq$ B o B $\subseteq$ A *(Uno dei due è contenuto nell'altro)*
- **Teorema del cammino bianco:** un vertice *v* è discendente di un vertice *u* SSE al momento della scoperta di *u*, il vertice *v* è raggiungibile da *u* tramite un cammino che contiene esclusivamente nodi bianchi *(questo teorema è un'applicazione del teorema delle parentesi)*.
- **Classificazione degli archi** in base al colore di arrivo:
  - WHITE: Tree-edge, arco appartenente alla foresta DFS, lati che portano a scoprire nuovi vertici.
  - GREY: Back-edge, archi che non appartengono alla foresta DFS, che vanno da un vertice *v* ad un antenato di *v* nell'albero DFS.
  - BLACK, vale solo per i grafi orientati:
    - Forward-edge, archi non appartenti alla foresta DFS che vanno da un vertice *u* ad un suo successore.
    - Cross-edge, tutti gli altri archi, posso identificarli guardando gli intervalli considerando il teorema delle parentesi.

ESERCIZI

CONTARE I VERTICI

- Possiamo contare in maniera *'bruta'* i vertici, ovvero incrementando un contatore ogni volta che ne scopriamo uno e lo coloriamo di grigio.

  ```
  DFS_Visit(G, u)
  u.col = GREY
  nv++

  for all w ∈ Adj[u]
      if w.col == WHITE
          DFS_Visit(G, w)
  ```

u.col = BLACK

- Per contare i vertici, oltre al metodo visto, possiamo aggiornare un contatore ogni volta che finiamo di esplorare un nodo aggiungendoci quanti nodi c'erano sotto di esso *(somma ricorsiva)*.

  DFS_Visit(G, u)
  u.col = GREY

  for all w $\in$ Adj[u]
     if w.col == WHITE
        k += DFS_Visit(G, w)

  u.col = BLACK
  Return k++ *//++ perché va contato anche u*

- Un terzo modo per contare i vertici si basa sull'utilizzo dei tempi; sappiamo che per ogni nodo ci salviamo due tempi, dunque facciamo due incrementi della variabile time per ogni vertice, di conseguenza il numero di nodi sarà pari a time/2.

  Quindi per vedere quanti nodi ci sono sotto un vertice *v* mi basta calcolare $\frac{v.f \space - \space v.d \space + \space 1}{2}$

  DFS(G)
  for all v $\in$ V
     v.col = WHITE
     v.π = NIL
  time = 0

  for all v $\in$ V
     if v.col == WHITE
        DFS_Visit(G, v)
        nv = (v.f - v.d + 1)/2

CONTARE LE CC DI UN GRAFO

Dato G = (V, E) non orientato, contare le sue componenti connesse.
Se ragioniamo su come è strutturato l'algoritmo DFS possiamo osservare che ogni volta che viene chiamata la DFS_Visit è come se venisse scelta una *'sorgente'* dalla quale scoprire il suo sotto grafo; di conseguenza ogni volta che tale procedura viene invocata, andiamo ad analizzare una nuova componente connessa.
(Per motivi di leggibilità e semplicità scriverò solo la procedura modificata, in esame vanno scritte entrambe).

DFS(G)
for all v $\in$ V
   v.col = WHITE
   v.π = NIL
time = 0
CC = 0

for all v $\in$ V
   if v.col == WHITE
      CC++
      DFS_Visit(G, v)
Return CC

CONTA ALBERI

Dato un grafo non orientato G = (V, E), calcolare quante sue CC sono degli alberi.

```
DFS(G)
for all v ∈ V
   v.col = WHITE
   v.π = NIL
nTree = 0

for all v ∈ V
   if v.col == WHITE
      Acyclic = TRUE
      DFS_Visit(G, v)
      if Acyclic
         nTree++

DFS_Visit(G, u)
u.col = GREY

for all w ∈ Adj[u].length
   if w.col == WHITE
      w.π = u
      DFS_Visit(G, w)
   else if w.col == GREY AND w != u.π
      Acyclic = FALSE

u.col = BLACK
```

ATTENZIONE: Notare che settiamo acyclic a false solo se il vertice considerato è diverso dal parent di *u*, questo perché nel caso dei grafi non orientati il parent sarà sempre presente nella lista di adiacenza dei nodi *(ma questo non significa che ci sia un ciclo)*.

ATTENZIONE: Nel caso di grafi orientati non ci dovremmo preoccupare del parent, ma occhio a ***non*** considerare gli archi neri come cicli; ci danno delle vie alternative ma non creano cicli.

ALTERNATIVA CONFRONTANDO ARCHI E NODI

```
DFS(G)
for all v ∈ V
   v.col = WHITE
nTree = 0

for all v ∈ V
   if v.col == WHITE
      nv = 0, ne = 0
      DFS_Visit(G, v)
      if ne/2 == nv - 1
         nTree++

DFS_Visit(G, u)
u.col = GREY
nv++
```

```
for all w ∈ Adj[u]
   ne++
   if w.col == WHITE
      DFS_Visit(G, w)

u.col = BLACK
```

CC COMPLETE

Dato un grafo non orientato G = (V, E) contare il numero di componenti connesse che siano dei grafi completi. *(Un grafo non orientato è completo quando |E| = (|V| * (|V| - 1))/2).*

```
DFS(G)
for all v ∈ V
   v.col = WHITE
nComp = 0

for all v ∈ V
   if v.col == WHITE
      nv = 0, ne = 0
      DFS_Visit(G, v)
      if ne/2 == ((nv - 1) * nv)\2
         nComp++

DFS_Visit(G, u)
u.col = GREY
nv++

for all w ∈ Adj[u]
   ne++
   if w.col == WHITE
      DFS_Visit(G, w)

u.col = BLACK
```

ETICHETTARE ARCHI

Dato un grafo orientato G = (V, E) stabilire, per ogni lato *(i, j)* ∈ E, di che tipo di lato si tratta.

```
DFS_Visit(G, u)
time++
u.d = time
u.col = GREY

for all w ∈ Adj[u]
   if w.col == WHITE
      print 'Tree-Edge'
      w.π = u
      DFS_Visit(G, w)
   else if w.col == GREY      print 'Back-Edge'
   else if u.d < w.d          print 'Forward-Tree'
   else                       print 'Cross-Edge'
```

u.col = BLACK
time++
u.f = time

ATTENZIONE: Nel caso dei grafi non orientati sappiamo che i casi di vertici BLACK non si verificheranno mai; tuttavia attenzione al ciclo for: non dobbiamo considerare il parent di u, altrimenti avremmo un arco etichettato come Back quando in realtà è Tree.

FORESTA CON K ALBERI

Dato un G = (V, E) non orientato e k > 0 stabilire se G è una foresta con esattamente k alberi.

DFS(G)
for all v $\in$ V
  v.col = WHITE
  v.π = NIL
time = 0
CC = 0

while all v $\in$ V **AND CC $\leq$ k AND Acyclic**
  if v.col == WHITE **AND CC < k**
    DFS_Visit(G, v)
    CC++
  else if v.col == WHITE
    CC++ *(se ho già sforato k non ha senso fare un'altra visita, incremento CC per dire che CC > k)*
if CC == k AND Acyclic
  Return TRUE
else
  Return FALSE

DFS_Visit(G, u)
time++
u.d = time
u.col = GREY

while all w $\in$ Adj[u] **- u.π AND Acyclic**
  if w.col == WHITE
    w.π = u
    DFS_Visit(G, w)
  else
    Acyclic = FALSE

u.col = BLACK
time++
u.f = time